APPENDIX A – 24/8 Update Method

```
if (prefix_length <= 24) {
   prefix_diff = 24 - prefix_length;
   prefix = ip_addr >> (32 - prefix_length);

   for (ui = 0; ui < (1 << prefix_diff); ui++) {
      // compute index into first level table T1_RIB
      t1_index = (prefix << prefix_diff) + ui;
      result[15:0] = T1_RIB[t1_index];

      if ( result[15] == 1'b0) {
         // the first bit in T1_RIB is 0
         old_prefix_length = result[5:0];
         if (old_prefix_length <= prefix_length) {
            // update with new next hop and prefix length
               T1_RIB[t1_index] = (next_hop <<6 +prefix_length)
               & 0x7FFF;
            }
         }
      else { // the first bit is 1
         for (uj = 0; uj < 256; uj++) {
            t2_index = result[14:6] <<8 +uj;
            t2_result[15:0] = T2_RIB[t2_index];
               if (t2_result[5:0] <= prefix_length) {
                  // assign next next hop and prefix length
                  T2_RIB[t2_index] = (next_hop << 6) + prefix_length;
                  }
            } // end of FOR loop
         } // end of ELSE loop
      } // end of FOR loop
   } // end of IF loop

else { // i.e., prefix_length > 24
      offset_size = 32 - prefix_length;
   prefix_diff = prefix_length - 24;
   t1_index = ip_addr[31:8];
   t1_result[15:0] = T1_RIB[t1_index];

   // find the end bit position in the ip address and compute offset
   offset = ip_addr[7:offset_size];
```

```
    if (t1_result[15] == 1'b0) { // the first bit is 0 in T1_RIB
        index = get_index(); // get a new index into T2_RIB
        // compute the begin & end address of the segment in T2_RIB
        segment_begin_addr = (index << 8) + offset;
        segment_end_addr = (index <<8) + offset +
        (1 << offset_size) - 1;

        // update T1_RIB with the next index and mark 1st bit to 1
        T1_RIB[t1_index] = (index << 6) | 0x8000;

        for (uj = 0; uj < 256; uj++) {
            t2_index = (index << 8) + uj;
            if ( (t2_index >= segment_begin_addr) &&
                (t2_index <= segment_end_addr) ) {
                // this is the segment needs to be updated
                // with new NH and prefix length information.
                T2_RIB[t2_index]=(next_hop << 6)+prefix_length;
            }
            else { // update unmatching segment with old NH & PL
                  T2_RIB[t2_index] = t1_result[15:0];
                }
            } // end of FOR loop
        } // end of IF loop
    else { // the first bit in T1_RIB is 1
index = t1_result[14:6];
        // compute the begin & end address of the segment in T2_RIB
        segment_begin_addr = (index << 8) + offset;
        segment_end_addr = (index <<8) + offset +
        (1 << offset_size) - 1;

        for (uj = 0; uj < 256; uj++) {
            t2_index = (index << 8) + uj;
            if ( (t2_index >= segment_begin_addr) &&
               (t2_index <= segment_end_addr) ) {
               // this is the segment matching the prefix
               old_prefix_length = T2_RIB[t2_index] & 0x003F;
               if (old_prefix_length <= prefix_length) {
                  // need to update wit new NH and PL
                  T2_RIB[t2_index]=(next_hop<<6)+prefix_length;
               }
            } // end of IF loop
        } // end of FOR loop
    } // end of ELSE loop
} // end of ELSE loop
```

APPENDIX B -- Pseudo Code of 24/8c Route Lookup Method

```
// get segment and offsets from IP address
  segment[17:0] = ip_addr[31:14];
  offset1[5:0] = ip_addr[13:8];
  offset2[7:0] = ip_addr[7:0];

  t1_result[95:0] = T1_RIB[segment];

// compute the number of ones in the bit map
  all_ones = compute_num_ones_in_bitmap(t1_result[95:0]);
  K = 95 - offset1[5:0];
  leading_ones = compute_num_ones_in_bitmap(t1_result[95:K]);
  if (all_ones <= 2) {
if (leading_ones == 1)
nhpl[15:0] = t1_result[31:16];
else
nhpl[15:0] = t1_result[15:0];
  }

  else { // the total number of one's is more than 2
      address = t1_result[31:0];
      current_address = address + leading_ones - 1;

      // retrieve nhpl value stored in the address
      nhpl[15:0] = *current_address;
  }

  if ( (nhpl[15] == 1'b0) { // first bit in NHPL is 0
    next_hop = nhpl[14:6];
  }
  else { // first bit in NHPL is 1
    index_t2 = nhpl[14:0];
    t2_index = index_t2 << 8 + offset2;
    t2_result[15:0] = T2_RIB[t2_index];
    next_hop = t2_result[15:6];
  }
```

APPENDIX C -- Pseudo Code of 24/8c Route Update Method

```
segment = ((ip_addr)>>14); // get the first 18 significant bits
new_nhpl = (next_hop<<6) + prefix_length; // compute new NHPL

if (prefix_length <= 18) {
  // NO need to change the bitmap
  // only need to update NHPL array as needed
  pl_diff = 18 - prefix_length;
  t1_base = (segment>>pl_diff)<<pl_diff;

  for (ti = 0; ti < (1<<pl_diff); ti++) {
    t1_index = t1_base + ti;
    t1_result = &(T1_RIB[t1_index]);

    bmp = t1_result->bmp; // bmp stores 64 bitmap t1_entry[95:32];
    all_ones = count_leading_ones(bmp, 63);

    for (i = 1; i <= all_ones; i++) {
    // walk throught the NHPL array and make changes as needed
    // get the old nhpl
    // i indicates the leading ones
    nhpl = get_current_nhpl(t1_result, all_ones, i);

    if ( (nhpl & 0x8000) == 0) { // marker bit is 0
      old_pl = (nhpl & 0x003F);
      if ( (old_pl <= prefix_length) && (nhpl != new_nhpl) ) {
        // update T1_RIB with new nhpl
        replace_current_nhpl(t1_result, all_ones, i, new_nhpl);
      }
    }
    else { // marker bit is 1
      t2_index = ( (nhpl & 0x7FFF) << 8); // times 256
      begin_index = t2_index;
      end_index = t2_index + 255;

      update_t2_rib(begin_index, end_index, prefix_length, next_hop);
    }
   } // end of for (i = 1; i <= all_ones; ...)
  } // end of for (ti = 0; ...)
} // end of if (prefix_length <= 18)

else if (prefix_length <= 24) {  // 18 < prefix_length <= 24
  // get the next 6 bits after the significant 18 bits as offset1
  offset1 = ((ip_addr<<18)>>26); // 6 bits
  pl_diff = 24 - prefix_length;

  t1_result = &(T1_RIB[segment]);
  t1_bmp_base = ( (offset1 >> pl_diff) << pl_diff);

  for (ti = 0; ti < (1 << pl_diff); ti++) {
    pos = t1_bmp_base + ti; // get the bit position

    // count the number of all ones and leading ones
    bmp = t1_result->bmp;
    all_ones = count_leading_ones(bmp, 63);
```

```
    leading_ones = count_leading_ones(bmp, pos);

    // get the old prefix info
    nhpl = get_current_nhpl(t1_result, all_ones, leading_ones);

    if ( ( (nhpl & 0x8000) == 0) && (nhpl != new_nhpl) ) {
    // the marker bit is 0 and new nhpl is NOT equal to the old nhpl
    // in this case, we may need to change bitmap and nhpls
    old_pl = (nhpl & 0x003F);
    if (old_pl <= prefix_length) {
      // update with new nhpl and update bitmap
      cb = ((bmp << pos) >> 63); // get current bit

      if (pos == 0) { // this is the first bit
        // get the next bit
        nb = ((bmp << (pos + 1) ) >> 63);

        if (nb == 0) { // change the bitmap "10x" --> "11x"
          t1_result->bmp |= 0xC000000000000000LL;
          // since pos == 0, leading_ones has to be 1
          insert_before_pos_nhpl(t1_result, all_ones, 1, new_nhpl);
        }

        else { // nb == 1
          next_nhpl = get_next_nhpl(t1_result,all_ones,leading_ones);
          if ( next_nhpl == new_nhpl) {
            // change the bitmap "11x" --> "10x", set next bit to 0
            bits_mask = (1LL<<(62-pos));
            t1_result->bmp &= ~bits_mask;

            delete_current_nhpl(t1_result, all_ones, leading_ones);
          }
          else { // no need to change the bitmap "11x"
            replace_current_nhpl(t1_result, all_ones, 1, new_nhpl);
          }
        }
      } // end of if (pos == 0)

      else if (pos == 63) { // end bit
        if (cb == 0) { // current bit is 0
          // change bit map from "x0" --> "x1"
          t1_result->bmp = (t1_result->bmp | 0x0000000000000001LL);

          insert_after_pos_nhpl(t1_result, all_ones, leading_ones,
                                new_nhpl);
        }
        else { // cb == 1
          // get previous nhpl
          prev_nhpl = get_prev_nhpl(t1_result,all_ones,leading_ones);

          if (new_nhpl == prev_nhpl) { // P C --> P
            // change bit map from "x1" --> "x0"
            t1_result->bmp = (t1_result->bmp & 0xFFFFFFFFFFFFFFFELL);

            delete_current_nhpl(t1_result, all_ones, leading_ones);
          }
```

```
      else { // new_nhpl != prev_nhpl, P C -- > P N
        // no need to change bitmap
        replace_current_nhpl(t1_result, all_ones, leading_ones,
                        new_nhpl);
      }
    }

  } // end of if (pos == 63)

  else { // this is a middle bit
    // get the next bit
    nb = ((bmp << (pos + 1) ) >> 63);

    if ( (cb == 0) && (nb == 0) ) {  // bmp: "x00x"
      // change bitmap "x00x" --> "x11x"
      // 0...0110..0 where the first 1 starts at pos
      bits_mask = ( (1LL<<(63-pos)) | (1LL<<(62-pos)) );
      t1_result->bmp |= bits_mask;

      insert_after_pos_dup_nhpl(t1_result, all_ones,
                          leading_ones, new_nhpl);
    } // end of (cb == 0, nb == 0)

    else if ( (cb == 0) && (nb == 1) ) { // bmp: "x01x"
      // get next nhpl
      next_nhpl = get_next_nhpl(t1_result,all_ones,leading_ones);

      if (next_nhpl != new_nhpl) { // F != N
        // change bitmap "x01x" --> "x11x"
        bits_mask = ( (1LL<<(63-pos)) | (1LL<<(62-pos)) );
        t1_result->bmp |= bits_mask;

        insert_after_pos_nhpl (t1_result, all_ones,
                          leading_ones, new_nhpl);
      }

      else { // F == N
        // change bitmap "x01x" --> "x10x"
        bits_mask = (1LL<<(63-pos));
        // set pos bit to 1
        t1_result->bmp |= bits_mask;
        // set the next bit to 0
        bits_mask = (1LL<<(62-pos));
        t1_result->bmp &= ~bits_mask;

      // NO need to change NHPL array
      }
    }

    else if ( (cb == 1) && (nb == 0) ) { // bmp: "x10x"
      prev_nhpl = get_prev_nhpl(t1_result,all_ones,leading_ones);

      if (prev_nhpl != new_nhpl) { // P != N
        // change bit map "x10x" --> "x11x"
        bits_mask = ( (1LL<<(63-pos)) | (1LL<<(62-pos)) );
        t1_result->bmp |= bits_mask;
```

```
        insert_before_pos_nhpl (t1_result, all_ones,
                        leading_ones, new_nhpl);
      }
      else { // P == N
        // change bit map "x10x" --> "x01x"
        // set pos bit to 0
        bits_mask = (1LL<<(63-pos));
        t1_result->bmp &= (~bits_mask);
        // set the next bit to 1
        bits_mask = (1LL<<(62-pos));
        t1_result->bmp |= bits_mask;

        // NO need to change NHPL array
      }
    }

    else if ( (cb == 1) && (nb == 1) ) { // bmp: "x11x"
      next_nhpl = get_next_nhpl(t1_result,all_ones,leading_ones);
      prev_nhpl = get_prev_nhpl(t1_result,all_ones,leading_ones);

      if (next_nhpl == new_nhpl) { // P C F --> P F
        // change the bitmap "x11x" --> "x10x"
        // set the next bit to 0
        bits_mask = (1LL<<(62-pos));
        t1_result->bmp &= ~bits_mask;

        delete_current_nhpl (t1_result, all_ones, leading_ones);
      }
      else if (prev_nhpl == new_nhpl) { // P C F --> P F
        // change the bitmap "x11x" --> "x01x"
        // set the current bit to 0
        bits_mask = (1LL<<(63-pos));
        t1_result->bmp &= ~bits_mask;

        delete_current_nhpl (t1_result, all_ones, leading_ones);
      }
      else { // P C F --> P N F
        // no need to change the bitmap "x11x"
        replace_current_nhpl (t1_result, all_ones,
                        leading_ones, new_nhpl);
      }
    } // end of if ( (cb == 1) && (nb == 1))
  } // End of else this is a middle bit
} // end of if (old_pl <= prefix_length)
} // end of if ( ( (nhpl & 0x8000) == 0) && (nhpl != new_nhpl) )

else if ( (nhpl & 0x8000) != 0) { // the marker bit is 1
  // NO need to change bitmap pattern
  // only need to update T2_RIB
  index = (nhpl & 0x7FFF);
  t2_index = (index << 8);
  begin_index = t2_index;
  end_index = t2_index + 255;

  update_t2_rib(begin_index, end_index, prefix_length, next_hop);

} // end of if ( (nhpl & 0x8000) != 0)
```

```
    } // end of for (ti = 0; ti < (1<<pl_diff); ...)
  } // end of if (prefix_length <= 24)

  else { // prefix_length > 24
    offset1 = ((ip_addr<<18)>>26);
    pl_diff = prefix_length - 24;
    // get the significant bits after the first 24 significant bits
    offset_temp =( (ip_addr<<24) >> (32 - pl_diff) );
    offset_t2 =( offset_temp << (32 - prefix_length) );
    segment_size = (1 << (32 - prefix_length));

    t1_result = &(T1_RIB[segment]);

    bmp = t1_result->bmp;
    pos = offset1;

    // count the number of all ones and leading ones
    all_ones = count_leading_ones(bmp, 63);
    leading_ones = count_leading_ones(bmp, pos);

    // get the old prefix info
    nhpl = get_current_nhpl(t1_result, all_ones, leading_ones);

    if ( (nhpl & 0x8000) == 0) { // the marker bit is 0
    // DO need to change bitmap
      t2_index = get_t2_index();
      begin_index = (t2_index << 8);
      end_index = begin_index + 255;

cb = ((bmp << pos) >> 63);  // get current bit

      // marker the first bit to 1
      t1_new_nhpl = (t2_index | 0x8000);

      if (pos != 63) { // this is NOT the end bit
        // get current bit and next bit
        nb = ((bmp << (pos + 1) ) >> 63);

        // for all cases, we need to change bitmap to "xx" -- >"11"
        bits_mask = ( (1LL<<(63-pos)) | (1LL<<(62-pos)) );
        t1_result->bmp |= bits_mask;

        if ( (cb == 0) && (nb == 0) ) { // bitmap: "00x" -- > "11x"
          insert_after_pos_dup_nhpl(t1_result, all_ones,
                          leading_ones, t1_new_nhpl);
        }

        else if ( (cb == 0) && (nb == 1) ) { // bitmap:"00x" -- > "11x"
        insert_after_pos_nhpl (t1_result, all_ones,
                        leading_ones, t1_new_nhpl);
        }

        else if ( (cb == 1) && (nb == 0) ) { // bitmap:"00x" -- > "11x"
          insert_before_pos_nhpl (t1_result, all_ones,
                          leading_ones, t1_new_nhpl);
        }
```

```
        else if ( (cb == 1) && (nb == 1) ) { // bitmap:"00x" -- > "11x"
          replace_current_nhpl (t1_result, all_ones,
                        leading_ones, t1_new_nhpl);
        }
      } // end of if (pos != 63)

      else { // end bit
        if (cb == 0) { // change bitmap: "x0" --> "x1"
            // change bit map from "x0" --> "x1"
            t1_result->bmp |= 0x0000000000000001LL;

            insert_after_pos_nhpl(t1_result, all_ones, leading_ones,
                          t1_new_nhpl);

        }
        else { // cb == 1
          // no need to change bitmap
          replace_current_nhpl (t1_result, all_ones,
                        leading_ones, t1_new_nhpl);
        }

      }

      // update T2_RIB with old nhpl for those nonmatching segment
      // update T2_RIB with new nhpl for those matching segment
      begin_segment_index = begin_index + offset_t2;
      end_segment_index = begin_segment_index + segment_size - 1;
      first_update_t2_rib(begin_index, end_index, begin_segment_index,
                    end_segment_index, nhpl, new_nhpl);
    }
    else { // the marker bit is 1
      // no need to change bitmap
      t2_index = (nhpl & 0x7FFF);
      begin_index = (t2_index << 8) + offset_t2;
      end_index = begin_index + segment_size - 1;

      update_t2_rib(begin_index, end_index, prefix_length, next_hop);
    }
  }
}
```

```
state o_lookup1 96
// field nhop1 o_lookup1[31:16]
// field nhop2 o_lookup1[15:0]
// field nhopaddr = o_lookup1[31:0]
state leading_ones 32
// state all_ones 32

interface       VAddr           32      core    out
interface       LSSize          5       core    out
interface       MemDataIn128    128     core    in
interface       MemDataIn16     16      core    in

opcode LOAD128 op2=0 CUST0
opcode ALLONES op2=1 CUST0
opcode GETNHOPADDRFROMT1 op2=2 CUST0
opcode INDEXFORT2 op2=3 CUST0
opcode GETNHOPFROMT2 op2=4 CUST0
opcode LEADONES op2=5 CUST0
opcode GETNHOPADDRFROMADDR op2=6 CUST0

iclass load1 {LOAD128} {out arr, in ars, in art} {out o_lookup1} {
        out VAddr, out LSSize, in MemDataIn128
}
//
// 128 bit load from memory. MemDataIn128[127:64] has the bit map
// MemDataIn128[63:32] stores either two next hops (each of which is
// is a 16 bit data or a 32 bit address pointing to a list of
// next hops.
//
reference LOAD128 {
  assign LSSize = 5'b10000;
  assign VAddr = ars + art;
  assign arr = MemDataIn128[63:32];
  assign o_lookup1 = {MemDataIn128[127:32]};
}
//
// loads from memory is a 2 cycle operation
//
schedule S1 {LOAD128} {def o_lookup1 2; def arr 2;}

iclass comp1 {ALLONES} {out arr} {in o_lookup1} {
}

// Count occurence of all ones in the 64 bit bitmap array
//
// Method is 32 1-bit adders at the first level
//              16 2-bit adders at the second level
//              8 3-bit adders at the third level
//              4 4-bit adders at the fourth level
//              2 5-bit adders at the fifth level
//              1 6-bit adder at the sixth level
//
// Reason for parallelism is to speed up performance of
// addition. Hardware implementation for multi-bit adders
```

```
// is implemted as ripple carry adders.
// In order to understand the number of stages in this
// implemetation
// 6 XOR stages at the RTL level.
// 15 CARRY STAGES (1+2+3+4+5)
// Totaling 21 stage design

reference ALLONES {

// first level description
  wire[1:0] w0_0;
  wire[1:0] w0_1;
  wire[1:0] w0_2;
  wire[1:0] w0_3;

  wire[1:0] w0_4;
  wire[1:0] w0_5;
  wire[1:0] w0_6;
  wire[1:0] w0_7;

  wire[1:0] w0_8;
  wire[1:0] w0_9;
  wire[1:0] w0_10;
  wire[1:0] w0_11;

  wire[1:0] w0_12;
  wire[1:0] w0_13;
  wire[1:0] w0_14;
  wire[1:0] w0_15;

  wire[1:0] w0_16;
  wire[1:0] w0_17;
  wire[1:0] w0_18;
  wire[1:0] w0_19;

  wire[1:0] w0_20;
  wire[1:0] w0_21;
  wire[1:0] w0_22;
  wire[1:0] w0_23;
  wire[1:0] w0_24;
  wire[1:0] w0_25;
  wire[1:0] w0_26;
  wire[1:0] w0_27;

  wire[1:0] w0_28;
  wire[1:0] w0_29;
  wire[1:0] w0_30;
  wire[1:0] w0_31;

// second level description
  wire[2:0] w1_0;
  wire[2:0] w1_1;
  wire[2:0] w1_2;
  wire[2:0] w1_3;

  wire[2:0] w1_4;
  wire[2:0] w1_5;
```

```
  wire[2:0] w1_6;
  wire[2:0] w1_7;

  wire[2:0] w1_8;
  wire[2:0] w1_9;
  wire[2:0] w1_10;
  wire[2:0] w1_11;

  wire[2:0] w1_12;
  wire[2:0] w1_13;
  wire[2:0] w1_14;
  wire[2:0] w1_15;

// third level description
  wire[3:0] w2_0;
  wire[3:0] w2_1;
  wire[3:0] w2_2;
  wire[3:0] w2_3;

  wire[3:0] w2_4;
  wire[3:0] w2_5;
  wire[3:0] w2_6;
  wire[3:0] w2_7;

// fourth level description
  wire[4:0] w3_0;
  wire[4:0] w3_1;
  wire[4:0] w3_2;
  wire[4:0] w3_3;

// fifth level description
  wire[5:0] w4_0;
  wire[5:0] w4_1;
  assign  w0_0 = o_lookup1[95]+o_lookup1[94];
  assign  w0_1 = o_lookup1[93]+o_lookup1[92];
  assign  w0_2 = o_lookup1[91]+o_lookup1[90];
  assign  w0_3 = o_lookup1[89]+o_lookup1[88];

  assign  w0_4 = o_lookup1[87]+o_lookup1[86];
  assign  w0_5 = o_lookup1[85]+o_lookup1[84];
  assign  w0_6 = o_lookup1[83]+o_lookup1[82];
  assign  w0_7 = o_lookup1[81]+o_lookup1[80];

  assign  w0_8 =  o_lookup1[79]+o_lookup1[78];
  assign  w0_9 =  o_lookup1[77]+o_lookup1[76];
  assign  w0_10 = o_lookup1[75]+o_lookup1[74];
  assign  w0_11 = o_lookup1[73]+o_lookup1[72];

  assign  w0_12 = o_lookup1[71]+o_lookup1[70];
  assign  w0_13 = o_lookup1[69]+o_lookup1[68];
  assign  w0_14 = o_lookup1[67]+o_lookup1[66];
  assign  w0_15 = o_lookup1[65]+o_lookup1[64];

  assign  w0_16 = o_lookup1[63]+o_lookup1[62];
  assign  w0_17 = o_lookup1[61]+o_lookup1[60];
  assign  w0_18 = o_lookup1[59]+o_lookup1[58];
  assign  w0_19 = o_lookup1[57]+o_lookup1[56];
```

```
  assign  w0_20 = o_lookup1[55]+o_lookup1[54];
  assign  w0_21 = o_lookup1[53]+o_lookup1[52];
  assign  w0_22 = o_lookup1[51]+o_lookup1[50];
  assign  w0_23 = o_lookup1[49]+o_lookup1[48];
  assign  w0_24 = o_lookup1[47]+o_lookup1[46];
  assign  w0_25 = o_lookup1[45]+o_lookup1[44];
  assign  w0_26 = o_lookup1[43]+o_lookup1[42];
  assign  w0_27 = o_lookup1[41]+o_lookup1[40];

  assign  w0_28 = o_lookup1[39]+o_lookup1[38];
  assign  w0_29 = o_lookup1[37]+o_lookup1[36];
  assign  w0_30 = o_lookup1[35]+o_lookup1[34];
  assign  w0_31 = o_lookup1[33]+o_lookup1[32];

// second level description

  assign  w1_0 = w0_0+w0_1;
  assign  w1_1 = w0_2+w0_3;
  assign  w1_2 = w0_4+w0_5;
  assign  w1_3 = w0_6+w0_7;

  assign  w1_4 = w0_8+w0_9;
  assign  w1_5 = w0_10+w0_11;
  assign  w1_6 = w0_12+w0_13;
  assign  w1_7 = w0_14+w0_15;

  assign  w1_8 = w0_16+w0_17;
  assign  w1_9 = w0_18+w0_19;
  assign  w1_10 = w0_20+w0_21;
  assign  w1_11 = w0_22+w0_23;

  assign  w1_12 = w0_24+w0_25;
  assign  w1_13 = w0_26+w0_27;
  assign  w1_14 = w0_28+w0_29;
  assign  w1_15 = w0_30+w0_31;
// third level description

  assign  w2_0 = w1_0+w1_1;
  assign  w2_1 = w1_2+w1_3;
  assign  w2_2 = w1_4+w1_5;
  assign  w2_3 = w1_6+w1_7;

  assign  w2_4 = w1_8+w1_9;
  assign  w2_5 = w1_10+w1_11;
  assign  w2_6 = w1_12+w1_13;
  assign  w2_7 = w1_14+w1_15;

// fourth level description

  assign  w3_0 = w2_0+w2_1;
  assign  w3_1 = w2_2+w2_3;
  assign  w3_2 = w2_4+w2_5;
  assign  w3_3 = w2_6+w2_7;

// fifth level description
```

```
  assign  w4_0 = w3_0+w3_1;
  assign  w4_1 = w3_2+w3_3;

// sixth level description

  assign arr = w4_0+w4_1;
  // assign all_ones = w4_0+w4_1;
}
iclass comp1_1 {LEADONES} {out arr, in ars} {in o_lookup1, out leading_ones} {
}
// Method :
//
// Step 1 : result_tmp = o_lookup1>>(63-Value(ars))
// Step 2 : result = count_ones (result_tmp)
//
// First step is implemented as o_lookup1>>~ars[6:0]
//
// Count occurence of all ones in the 64 bit bitmap array
// Method is 32 1-bit adders at the first level
//              16 2-bit adders at the second level
//              8 3-bit adders at the third level
//              4 4-bit adders at the fourth level
//              2 5-bit adders at the fifth level
//              1 6-bit adder at the sixth level
//
// Reason for parallelism is to speed up performance of
// addition. Hardware implementation for multi-bit adders
// is implemted as ripple carry adders.
// In order to understand the number of stages in this
// implemetation
// 6 XOR stages at the RTL level.
// 15 CARRY STAGES (1+2+3+4+5)
// Totaling 21 stage design
reference LEADONES {

  wire[1:0] spw0_0;
  wire[1:0] spw0_1;
  wire[1:0] spw0_2;
  wire[1:0] spw0_3;

  wire[1:0] spw0_4;
  wire[1:0] spw0_5;
  wire[1:0] spw0_6;
  wire[1:0] spw0_7;

  wire[1:0] spw0_8;
  wire[1:0] spw0_9;
  wire[1:0] spw0_10;
  wire[1:0] spw0_11;

  wire[1:0] spw0_12;
  wire[1:0] spw0_13;
  wire[1:0] spw0_14;
  wire[1:0] spw0_15;

  wire[1:0] spw0_16;
  wire[1:0] spw0_17;
```

```
  wire[1:0] spw0_18;
  wire[1:0] spw0_19;

  wire[1:0] spw0_20;
  wire[1:0] spw0_21;
  wire[1:0] spw0_22;
  wire[1:0] spw0_23;
  wire[1:0] spw0_24;
  wire[1:0] spw0_25;
  wire[1:0] spw0_26;
  wire[1:0] spw0_27;

  wire[1:0] spw0_28;
  wire[1:0] spw0_29;
  wire[1:0] spw0_30;
  wire[1:0] spw0_31;

// second level description

  wire[2:0] spw1_0;
  wire[2:0] spw1_1;
  wire[2:0] spw1_2;
  wire[2:0] spw1_3;

  wire[2:0] spw1_4;
  wire[2:0] spw1_5;
  wire[2:0] spw1_6;
  wire[2:0] spw1_7;

  wire[2:0] spw1_8;
  wire[2:0] spw1_9;
  wire[2:0] spw1_10;
  wire[2:0] spw1_11;

  wire[2:0] spw1_12;
  wire[2:0] spw1_13;
  wire[2:0] spw1_14;
  wire[2:0] spw1_15;
  wire[3:0] spw2_0;
  wire[3:0] spw2_1;
  wire[3:0] spw2_2;
  wire[3:0] spw2_3;

  wire[3:0] spw2_4;
  wire[3:0] spw2_5;
  wire[3:0] spw2_6;
  wire[3:0] spw2_7;

// fourth level description

  wire[4:0] spw3_0;
  wire[4:0] spw3_1;
  wire[4:0] spw3_2;
  wire[4:0] spw3_3;

// fifth level description
```

```
  wire[5:0] spw4_0;
  wire[5:0] spw4_1;

  wire[6:0] result;

  wire[95:0] lookup1_sp;

  assign lookup1_sp = (o_lookup1>>(~ars[5:0]));
  assign  spw0_0 = lookup1_sp[95]+lookup1_sp[94];
  assign  spw0_1 = lookup1_sp[93]+lookup1_sp[92];
  assign  spw0_2 = lookup1_sp[91]+lookup1_sp[90];
  assign  spw0_3 = lookup1_sp[89]+lookup1_sp[88];

  assign  spw0_4 = lookup1_sp[87]+lookup1_sp[86];
  assign  spw0_5 = lookup1_sp[85]+lookup1_sp[84];
  assign  spw0_6 = lookup1_sp[83]+lookup1_sp[82];
  assign  spw0_7 = lookup1_sp[81]+lookup1_sp[80];

  assign  spw0_8 =  lookup1_sp[79]+lookup1_sp[78];
  assign  spw0_9 =  lookup1_sp[77]+lookup1_sp[76];
  assign  spw0_10 = lookup1_sp[75]+lookup1_sp[74];
  assign  spw0_11 = lookup1_sp[73]+lookup1_sp[72];

  assign  spw0_12 = lookup1_sp[71]+lookup1_sp[70];
  assign  spw0_13 = lookup1_sp[69]+lookup1_sp[68];
  assign  spw0_14 = lookup1_sp[67]+lookup1_sp[66];
  assign  spw0_15 = lookup1_sp[65]+lookup1_sp[64];

  assign  spw0_16 = lookup1_sp[63]+lookup1_sp[62];
  assign  spw0_17 = lookup1_sp[61]+lookup1_sp[60];
  assign  spw0_18 = lookup1_sp[59]+lookup1_sp[58];
  assign  spw0_19 = lookup1_sp[57]+lookup1_sp[56];

  assign  spw0_20 = lookup1_sp[55]+lookup1_sp[54];
  assign  spw0_21 = lookup1_sp[53]+lookup1_sp[52];
  assign  spw0_22 = lookup1_sp[51]+lookup1_sp[50];
  assign  spw0_23 = lookup1_sp[49]+lookup1_sp[48];
  assign  spw0_24 = lookup1_sp[47]+lookup1_sp[46];
  assign  spw0_25 = lookup1_sp[45]+lookup1_sp[44];
  assign  spw0_26 = lookup1_sp[43]+lookup1_sp[42];
  assign  spw0_27 = lookup1_sp[41]+lookup1_sp[40];

  assign  spw0_28 = lookup1_sp[39]+lookup1_sp[38];
  assign  spw0_29 = lookup1_sp[37]+lookup1_sp[36];
  assign  spw0_30 = lookup1_sp[35]+lookup1_sp[34];
  assign  spw0_31 = lookup1_sp[33]+lookup1_sp[32];

// second level description

  assign  spw1_0 = spw0_0+spw0_1;
  assign  spw1_1 = spw0_2+spw0_3;
  assign  spw1_2 = spw0_4+spw0_5;
  assign  spw1_3 = spw0_6+spw0_7;

  assign  spw1_4 = spw0_8+spw0_9;
  assign  spw1_5 = spw0_10+spw0_11;
  assign  spw1_6 = spw0_12+spw0_13;
```

```
  assign  spw1_7 = spw0_14+spw0_15;

  assign  spw1_8 = spw0_16+spw0_17;
  assign  spw1_9 = spw0_18+spw0_19;
  assign  spw1_10 = spw0_20+spw0_21;
  assign  spw1_11 = spw0_22+spw0_23;

  assign  spw1_12 = spw0_24+spw0_25;
  assign  spw1_13 = spw0_26+spw0_27;
  assign  spw1_14 = spw0_28+spw0_29;
  assign  spw1_15 = spw0_30+spw0_31;
// third level description

  assign  spw2_0 = spw1_0+spw1_1;
  assign  spw2_1 = spw1_2+spw1_3;
  assign  spw2_2 = spw1_4+spw1_5;
  assign  spw2_3 = spw1_6+spw1_7;

  assign  spw2_4 = spw1_8+spw1_9;
  assign  spw2_5 = spw1_10+spw1_11;
  assign  spw2_6 = spw1_12+spw1_13;
  assign  spw2_7 = spw1_14+spw1_15;

// fourth level description

  assign  spw3_0 = spw2_0+spw2_1;
  assign  spw3_1 = spw2_2+spw2_3;
  assign  spw3_2 = spw2_4+spw2_5;
  assign  spw3_3 = spw2_6+spw2_7;

// fifth level description

  assign  spw4_0 = spw3_0+spw3_1;
  assign  spw4_1 = spw3_2+spw3_3;

// sixth level description

  assign leading_ones = spw4_0+spw4_1;

  assign result = {spw4_0+spw4_1-1};
  assign arr = {result[6:0],1'b0}; // 2*(leading_ones-1)
                                   // 2*(leading_ones-1) gives the
                                   // result used by GETNHOPADDR
}
// schedule S2 {ALLONES} {def all_ones 1; def arr 1;}
schedule S2 {ALLONES} {def arr 1;}
schedule S2L {LEADONES} {def leading_ones 1; def arr 1;}

iclass comp2 {GETNHOPADDRFROMT1} {out arr} {in leading_ones, in o_lookup1} {}
//
// If leading_ones == 1 then return o_lookup1[31:16] otherwise
// return o_lookup1[15:0]
//
reference GETNHOPADDRFROMT1 {
  assign arr = (leading_ones == 1) ? o_lookup1[31:16] : o_lookup1[15:0] ;
}
```

```
schedule S3 {GETNHOPADDRFROMT1} {def arr 1;}

iclass comp2_1 {GETNHOPADDRFROMADDR} {in ars, in art, out arr} {} {
  out VAddr, out LSSize, in MemDataIn16
}
//
// 16bit load to get nhop from a list
// ars is the 32 bit base address stored in o_lookup1[63:32]
// art is the 16 bit offset from the base address which is
// computed as 2*(leading_ones - 1)
reference GETNHOPADDRFROMADDR {

  assign LSSize = 5'b00010;
  assign VAddr = ars[31:0] + art[15:0];
  assign arr = MemDataIn16;
}
//
//
schedule S3_1 {GETNHOPADDRFROMADDR} {def arr 2;}
//
iclass load2_sh {GETNHOPFROMT2} {out arr, in ars, in art} {} {
        out VAddr, out LSSize, in MemDataIn16
}
//
// 16 bit load to get nhop from T2_RIB
// ars is the base address &(T2_RIB[0])
// art is the offset which is computed from INDEXFORT2
//
reference GETNHOPFROMT2 {
  assign LSSize = 5'b00010;
  assign VAddr = ars[31:0] + art[15:0];
  assign arr = {6'b0,MemDataIn16[15:6]};
}
//
//
schedule S4_S {GETNHOPFROMT2} {def arr 2;}

iclass index {INDEXFORT2} {out arr, in ars, in art} {} {
}
//
// Table2 Index = 2*(nhop[14:0]*256) + &(T2_RIB[0])
//
reference INDEXFORT2 {
  assign arr = {{{ars[7:0],8'b0}+art[15:0]}<<1};
}
//
schedule S6 {INDEXFORT2} {def arr 1;}
```

APPENDIX E -- A Sample of C Code for Route Lookup with New Extended Instructions

```
static unsigned int rt_lookup_tie (unsigned int ip_addr) {
  // get the most significant 18 bits from IP address
  // and appended with 4 zeros since in TIE data is arranged by
  // byte while in T1_RIB by 128 bits.
  unsigned int offset = ((ip_addr>>14)<<4);
  unsigned short int t1offset;
  unsigned int nhpladdr, all_ones;
  unsigned int bytepos_leading_ones, nhpl;
  static T1_Entry *t1_base = &(T1_RIB[0]); // get T1_RIB base address
  static unsigned short *t2_base = &(T2_RIB[0]); // T2_RIB base address

  nhpladdr = LOAD128 (t1_base, offset); // load the 128 bit from T1_RIB
  t1offset = ((ip_addr<<18)>>26);  // get the middle 6 bits in ip_addr
  all_ones = ALLONES ();
  bytepos_leading_ones = LEADONES (t1offset);

  if (all_ones <= 2)
    nhpl = GETNHOPADDRFROMT1 ();
  else
    nhpl = GETNHOPADDRFROMADDR (bytepos_leading_ones, nhpladdr);

  if ((nhpl >>15) == 0) { // get the marker bit
    return nhpl>>6;
  }
  else {
    unsigned short int t2offset = ((ip_addr<<24)>>24);
    unsigned int t2segment_n_offset = INDEXFORT2 (nhpl,t2offset);
    return GETNHOPFROMT2 (t2_base, t2segment_n_offset);
  }
}
```

APPENDIX F -- Cycle Count of Route Lookup Function

```
00750000 :      40057504 :      <rt_lookup_tie>:
00750000 :      40057504 :      6c1004  entry   a1, 32
00250000 :      40057507 :      d520    mov.n   a5, a2
00255313 :      40057509 :      14fd22  l32r    a4, 40056994
00250000 :      4005750c :      05e314  srli    a3, a5, 14
00258399 :      4005750f :      244200  l32i    a4, a4, 0
00250000 :      40057512 :      0c3311  slli    a3, a3, 4
01043576 :      40057515 :      034660  load128 a6, a4, a3
00500000 :      40057518 :      000461  allones a4
00250000 :      4005751b :      058345  extui   a3, a5, 8, 6
00250007 :      4005751e :      003265  leadones a2, a3
00250000 :      40057521 :      6f4305  bgeui   a4, 3,4005752a
00187163 :      40057524 :      000462  getnhopaddrfromt1 a4
00187163 :      40057527 :      600002  j       4005752d
00471684 :      4005752a :      062466  getnhopaddrfromaddr a2, a6, a4
00687163 :      4005752d :      04f314  srli    a3, a4, 15
00250006 :      40057530 :      cc34    bnez.n  a3, 40057538
00249821 :      40057532 :      046214  srli    a2, a4, 6
00249821 :      40057535 :      060000  retw
00000537 :      40057538 :      13fd18  l32r    a3, 40056998
00000179 :      4005753b :      050247  extui   a2, a5, 0, 8
00000179 :      4005753e :      8330    l32i.n  a3, a3, 0
00000185 :      40057540 :      024263  indexfort2   a2, a4, a2
00001115 :      40057543 :      023264  getnhopfromt2  a2, a3, a2
00000179 :      40057546 :      d10f    retw.n
total cycles in block "rt_lookup_tie": 6592490
```